A Method for Functional Testing of Media Synchronization Protocols

Makoto Yamada, Takanori Mori, Atsushi Fukada,
Akio Nakata and Teruo Higashino

Department of Informatics and Mathematical Science,
Graduate School of Engineering Science, Osaka University,
Toyonaka Osaka, 560-8531, Japan

Abstract. In this paper, we propose a functional testing method of media synchronization protocols, which control the synchronization between audio and movie, described in concurrent synchronous timed I/O automata. In order to trace all test sequences (I/O event sequences) with synchronization on the model, we need to execute each I/O event at an adequate timing which satisfies the whole timing constraint for all the given test sequences. However, the outputs are given from the IUT and uncontrollable. Also each output/synchronization timing may affect executable timing for its succeeding I/O events in the test sequences. In this paper, we propose a technique to derive a set of time intervals which make all the given test sequences executable, and propose a method for functional testing using the technique.

1 Introduction

Recently many multimedia systems, which make use of various information media such as audio, video, images and text, have been developed and utilized. Multimedia systems usually can be modeled as real-time systems and timing constraints on I/O events are imposed upon such systems in order to guarantee their QoS. Especially, multimedia synchronization protocols can be modeled as a concurrent model in which multiple real-time modules work cooperatively [4, 5]. Timed automata or timed Petri nets have been taken into account for specifying such real-time systems[1] and used for specifying multimedia systems[3, 6]. Functional testing[7], which tests whether a given IUT (Implementation Under Test) possesses the functions designated in its specification, is an effective way for improving the reliability of such multimedia systems.

In this paper, we propose a functional testing method for media synchronization protocols, which control the synchronization among audio and video media. Here, for simplicity of discussion, we assume that each media synchronization protocol consists of three modules, (1) a module receiving and playing out audio data, (2) a module receiving and playing out video data and (3) a module controlling the synchronization between the above two modules. The modules (1) and (2) receive audio and video data through the network. The control module (3) synchronizes audio and video every suitable intervals. As the testing items, we consider the following items, (a) whether the synchronization among audio and video media works properly and (b) whether the IUT

can execute an alternative operation when the IUT cannot receive data because of the delay in the network.

We assume that each module of the media synchronization protocol is described as a timed I/O automaton[2]. In the timed I/O automaton model, a clock and registers are available. And each timing constraint on state transitions consists of a logical product of linear inequalities composed of the variables that keep the values of the clock and registers. We generate test sequences that consist of a series of I/O events for each timed I/O automaton. However, the generated sequences may not be always executable because executable timing constraints may be imposed on state transitions. Therefore we need to check the executability of the test sequences, and if they are executable, we also need to find suitable values of the variables (such as I/O event timing) that enable the sequence execute. Generally each input to the IUT can be given at the specified timing. On the other hand, the timing of each output from the IUT is uncontrollable. Accordingly, when we check the traceability of a test sequence, it is desirable not only finding an input timing set that makes the whole test sequence executable but also finding feasible input/output interval set.

In [2], we proposed a method for checking the traceability of a state transition sequence on a single timed I/O automaton automatically. If the sequence is traceable, we can derive values of such parameters as input timing automatically. In this paper, combining this method and an idea of [4] for scheduling event timing on concurrent EFSMs, we propose a method for (1) checking the traceability of concurrent sequences of state transitions on concurrent timed I/O automata, in which multiple timed I/O automata work cooperatively, and for (2) deriving a series of intervals of I/O event timing that enables the concurrent sequences.

The rest of this paper is organized as follows. In Section 2, we show our concurrent timed I/O automata model. In Section 3, the traceability of concurrent state transition sequences is defined. In Section 4, we propose a method for generating test sequences and deriving a set of executable intervals if the sequences are executable. And in Section 5, application results of our method are shown. Section 6 concludes this paper.

2 Concurrent Timed I/O Automata

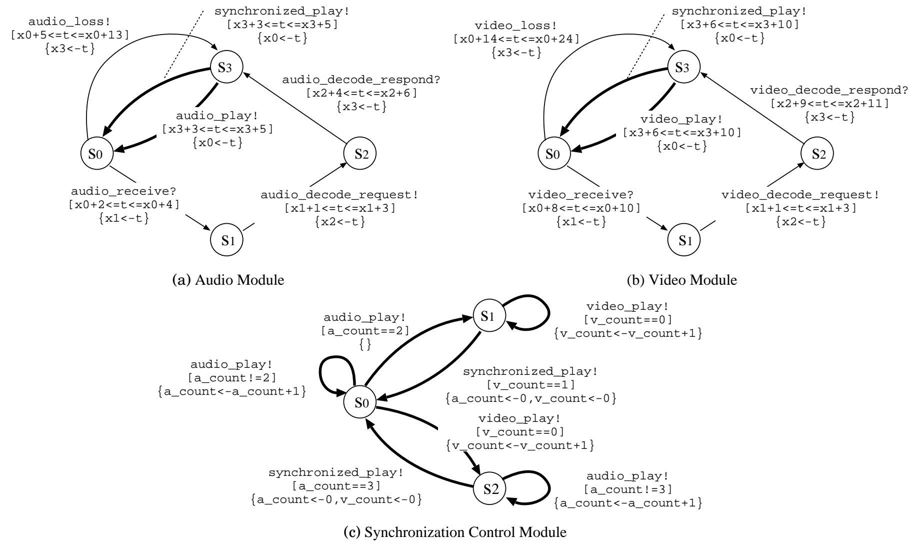
2.1 Timed I/O Automata

Definition 1. timed I/O automaton *is defined as a 10-tuple* $M =< S, A, I/Otype, t, V, Pred, Def, \delta, s_{init}, \{x_{1init}, x_{2init}, \ldots, x_{kinit}\} >$, *where*

- $S = \{s_0, s_1, \ldots, s_n\}$ *is a finite set of states.*
- *A is a finite set of I/O events.*
- $I/Otype = \{!, ?\} \cup \{?_{\mathbf{v}} | \mathbf{v} \in V\}$ *is a finite set of I/O types, where the symbols* ? *and* ! *represent input and output, respectively. The symbol* $?_{\mathbf{v}}$ *represents that the input value is assigned to the variable* **v**, *whose value may be used in the transition conditions of its succeeding events.*
- *t is a global clock variable which holds the current time.*
- $V = \{x_1, x_2, \ldots, x_k\}$ *is a set of variables.*

– *Pred is a set of transition conditions, each of which is a logical conjunction* $P[t, x_1, x_2, \ldots, x_k]$ *of linear inequalities consisting of variables.*
– Def *is a set of assignments. An assignment is a function which maps variables* $x_i \in V$ *to a linear expression* $f(t, \mathbf{v}, x_1, x_2, \ldots, x_k)$*, denoted by* $x_i \leftarrow f(t, \mathbf{v}, x_1, x_2, \ldots, x_k)$*.*
– δ *is a transition function.* $S \times A \times I/Otype \times V \rightarrow S \times V$*.*
– $s_{init} \in S$ *is the initial state of* M*.*
– $\{x_{1init}, x_{2init}, \ldots, x_{kinit}\}$ *is a set of the initial values of variables* $x_1, x_2, \ldots, x_k \in V$*.*

A transition on M *is denoted by* $s \xrightarrow{a\$[P]D} s'$*, where* $s, s' \in S$*,* $a \in A$*,* $\$ \in I/Otype$*,* $P \in Pred$ *and* $D \subseteq Def$*.* □



(a) Audio Module

(b) Video Module

(c) Synchronization Control Module

Fig. 1. Media Synchronization Protocol

For example, Fig.1 is a system which receives audio and video data from senders, synchronizes and plays out these media. The system is described as the concurrent timed I/O automata model whose formal semantics is described in Section 2.2. Modules (a) and (b) receive audio and video data, ask external decoders to decode them and play out them, respectively. They repeat these behavior. In this system, the fourth frame of audio and the second frame of video must be synchronized with module (c).

Intuitively, the semantics of our timed I/O automaton is as follows. At state s_0 in Fig.1(a), if the initial value of x_0 is zero, input event audio_receive? is executable between time 2 and 4 since the transition condition $x_0 + 2 \leq t \leq x_0 + 4$ becomes true. By executing this transition, module (a) moves to state s_1. In addition, the current value of clock t is assigned to variable x_1 according to the assignment $x_1 \leftarrow t$. If the execution

time of the input event is 3.5, then $x_1 = 3.5$ and output event audio_decode_request! becomes executable between time 4.5 and 6.5. If the output event is executed, module (a) moves to state s_2.

Module (a) receives an audio frame, asks a decoder to decode it and plays out it simultaneously with a video frame or independently. If module (a) cannot receive a frame (input event audio_receive? cannot be executed) at state s_0, module (a) assumes that the frame is lost, and it executes output event audio_loss! and moves to s_3.

In general, the behaviour of a timed I/O automaton is formally defined as a transition relation between *concrete states*. A concrete state is a pair of a control state and an value-assignment for all variable including the clock variable t. The transition relation between concrete states are defined as follows.

Definition 2. *A* value assignment *σ is a mapping from the set of state variables $V \cup \{t\}$ into the set of real-numbers R. For any value assignment σ and any non-negative real-value d, let $\sigma + d$ denote the same value assignment as σ except that the value of the clock variable t is increased by d (i.e. $(\sigma + d)(t) = \sigma(t) + d$). For any value assigment σ and any set of assignment statements $D \subseteq Def$, let σD denote the same value assignment as σ except that the assigned value of variables x_i appeared in the left hand of each assignment statements is the corresponding right hand expression, i.e. if $(x_i \leftarrow e_i) \in D$, then $\sigma D(x_i) = e_i$. We write $\sigma \models P$ if the predicate P holds when the value-assignment σ is applied.* □

Definition 3. *The transition relation between concrete states of a timed I/O automaton M is defined as the minimum relation derived by the following rules:*

- *For each transition $s \stackrel{a\$[P]D}{\longrightarrow} s'$,*
  - *for any value assignment σ and non-negative real-number d such that $(\sigma+d) \models P$, $(s,\sigma) \stackrel{d}{\longrightarrow} (s, \sigma + d)$ holds (transition by time passage).*
  - *if $\$ \neq ?_{\mathbf{v}}$, i.e. the action $a\$$ is not an input action with a variable, for any σ such that $\sigma \models P$, $(s,\sigma) \stackrel{a\$0}{\longrightarrow} (s, \sigma D)$ holds[1] (transition by I/O actions without input values)*
  - *if $\$ = ?_{\mathbf{v}}$, for any σ such that $\sigma \models P$, $(s,\sigma) \stackrel{a\$v}{\longrightarrow} (s, \sigma(D \cup \{\mathbf{v} \leftarrow v\}))$ (transition by an input action with an input value).* □

Note that our timed I/O automaton can simulate the classical Alur's timed automaton [1]. The details are shown in [2].

2.2 Concurrent Timed I/O Automata

In this paper, each media synchronization protocol is modeled as concurrent timed I/O automata which work cooperatively. Here, we assume that all the automata refer to the same clock variable t and that all the I/O events with the same name must be executed at the same time synchronously.

[1] For technical convenience, we attach a dummy value 0 for both input actions without input values and output actions.

For example, on the system shown in Fig.1, when synchronized_play! is executed on module (a), synchronized_play! is also executed on both modules (b) and (c). The transition condition for the synchronous events are the logical conjunction of the transition conditions corresponding to synchronized_play! on modules (a), (b) and (c). So, the synchronous event is executable if and only if the three transition conditions hold. Formally, it is defined as follows.

Definition 4. *A* concurrent timed I/O automaton $\mathcal{M}$ *is a finite vector of timed I/O automata* $(M_1, \ldots, M_k)$*, whose behavior is defined by the minimum transition relation between concrete states derived by the following rules:*

- *for any* $i \in \{1, \ldots k\}$*, if* $(s_i, \sigma_i) \xrightarrow{d} (s'_i, \sigma'_i)$*, then* $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k) \xrightarrow{d} ((s'_1, \ldots, s'_k), (\sigma'_1, \ldots, \sigma'_k)$ *holds (synchronized time passages).*
- *for any I/O action a\$, let I be the set of indices of timed I/O automata such that the transition* $(s_i, \sigma_i) \xrightarrow{a\$v} (s'_i, \sigma'_i)$ *is executable. Then* $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k)) \xrightarrow{a\$v} ((s'_1, \ldots, s'_k), (\sigma'_1, \ldots, \sigma'_k))$*, where* $s'_j = s_j$ *and* $\sigma'_j = \sigma_j$ *for* $j \notin I$ *(synchronized execution of the same I/O actions).* □

3 Executability of Sequences

In testing of real-time protocols, tester can usually control input timing to IUT. On the other hand, tester cannot control output timing from IUT. Each output timing may affect executable timing for its succeeding I/O events in sequences. Hence, it is desirable that we can find I/O event sequenses which can be executed whenever the IUT produces outputs. In this section, we define executability of sequences on our concurrent I/O timed automata.

3.1 Symbolic Trace

Since values of variables may be updated by executing transitions in a timed I/O automaton, we have to consider how to change the values of those variables for deciding executability of sequences. We translate values of variables and transition constraints in given sequences into expressions(symbolic traces) consisting of initial values of variables, execution time of transition or input values[2].

Formally symbolic traces are defined as follows.

Definition 5. *For a path* $\alpha = s_1 a_1 \$_1 s_2 \ldots s_k a_k \$_k s_{k+1}$ *on a timed I/O automaton M (each* $s_i, a_i, \$_i (i \in \{1, \ldots, k+1\})$ *can be repeated), sequence* $w = (a_1\$_1, t_1, P_1) \cdots (a_k\$_k, t_k, P_k)$ *is said to be a symbolic trace of* α*, if the following conditions hold.*

- $t_1, \ldots, t_k$ *are n different variables.*
- *Assume that Iin denotes the set of subscripts* $\{i \mid \$_i = ?_{v_i}\}$*.* $v_i (i \in Iin)$ *are n different variables.*
- *Each predicate* P_i *contains only the initial values of state variables V on M* $(x_{1init}, \ldots, x_{kinit})$*, input variables before i-th event* $(\{v_j \mid j \in Iin \cap \{1, \ldots, i\}\})$ *and variables* $(t_1, \ldots, t_i)$*.*

– *If M can execute a sequence $(s_1, \sigma_1) \xrightarrow{d_1} (s_1, \sigma'_1) \xrightarrow{a_1\$_1 v_1} (s_2, \sigma_2) \xrightarrow{d_2} \dots \xrightarrow{a_i\$_i v_i} (s_{i+1}, \sigma_{i+1})$, then $P_1 \wedge \cdots \wedge P_k$ is satisfied, where $\{v_i \mid \$_i = ?_{v_i}\}$ are input values v_i and $t_j (j \in \{1, \dots, i\})$ are execution time of event a_j ($t_j = \Sigma_{m=1}^{j} d_m$). In this case, we say that the execution sequence above* satisfies *the symbolic trace w.* □

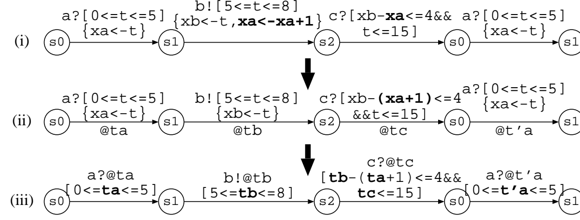


Fig. 2. Symbolic Trace

Let us construct a symbolic trace for a sequence shown in Fig.2 (i). First, we express each variable by expressions containing some of the preceding execution time, initial values of variables and input variables. Next, for the obtained sequence(ii), we prepare variables (t_a, t_b, t_c) for execution time of events and express constraints of transitions by using these variables. As a result we obtain a symbolic trace for the sequence $a?@t_a[0 \leq t_a \leq 5], b!@t_b[5 \leq t_b \leq 8], c?@t_c[t_b - (t_a + 1) \leq 4 \leq t_c \leq 15], a?@t'_a[0 \leq t'_a \leq 5]$.

We say that a symbolic trace w is traceable, if for some output timing there exists some input timing such that the succeeding sequence can be executed. We formalize the traceability as follows.

Definition 6. *For symbolic trace $w = (a_1\$_1, t_1, P_1) \cdots (a_k\$_k, t_k, P_k)$, $T(w)$ denotes a tuple of conditions $(P'_1, \dots, P'_k)$ for the initial values of variables $(x_{1init}, \dots, x_{kinit})$, execution times t_i of events $a_i\$_i$ $(1 \leq i \leq k)$ and input values v_i (if they exist) $(1 \leq i \leq k)$, where P'_i is equivalent to P_i or stricter condition than P_i $(P'_i \Rightarrow P_i)$. We call $T(w)$ as an* I/O timing interval. □

Intuitively, we say that a symbolic trace w is traceable by an I/O timing interval $T(w)$ if the corresponding action sequence of w is executable at any I/O timing which satisfies $T(w)$.

Definition 7. *A symbolic trace $w = (a_1\$_1, t_1, P_1) \dots (a_k\$_k, t_k, P_k)$ is* traceable *with respect to I/O timing interval $T(w) = (P'_1, \dots, P'_k)$, if for any solution $(t_1, \dots, t_k, \mathbf{v_1}, \dots, \mathbf{v_k})$ of the condition $P'_1 \wedge \cdots \wedge P'_k$, there exists some concrete trace $(s_1, \sigma_1) \xrightarrow{d_1} (s_1, \sigma'_1) \xrightarrow{a_1\$_1 v_1} (s_2, \sigma_2) \xrightarrow{d_2} \dots \xrightarrow{a_i\$_i v_i} (s_{i+1}, \sigma_{i+1})$ such that $d_1 = t_1$ and $d_i = t_i - t_{i-1}$ $(i \in \{2, \dots, k\})$.* □

3.2 Executability of Concurrent Sequences

The concrete trace of concurrent timed I/O automata is generally defined as a concrete trace of its composed automaton, as defined in Definition 4. Intuitively, we say that

such a concrete trace can *trace* the given set of symbolic traces if the *projection* of the trace to each consisting timed I/O automaton satisfies the corresponding symbolic trace. Moreover, if there exists such a concrete trace for a given set of symbolic traces, we say the set of symbolic traces is *traceable*. Formally, these notions are defined as follows.

Definition 8. *For sequences* $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k)) \xrightarrow{d_1} \xrightarrow{a_1\$_1} \cdots \xrightarrow{d_n} \xrightarrow{a_n\$_n} ((s'_1, \ldots, s'_k), (\sigma'_1, \ldots, \sigma'_k))$ *on concurrent timed I/O automata* $\mathcal{M} = (M_1, \ldots, M_k)$*,* $Proj_i(d_1a_1\$_1 \ldots d_na_n\$_n)$ *denotes a projection for timed I/O automaton* M_i*.*

$$Proj_i(d_1a_1\$_1\alpha) = \begin{cases} d_1a_1Proj_i(\alpha) & \text{if } a_1 \in A_i \\ Proj'_i(\alpha, d_1) & \text{otherwise.} \end{cases}$$

$$Proj_i(\epsilon) = \epsilon$$

$$Proj'_i(d_1a_1\$_1\alpha, d) = Proj_i((d + d_1)a_1\$_1\alpha))$$

$$Proj'_i(\epsilon, d) = d$$

Then, for a given set of symbolic traces $w_1, \ldots, w_k$ *of a concurrent timed I/O automaton* $\mathcal{M} = (M_1, \ldots, M_k)$ *and a concrete trace* α *of* $\mathcal{M}$*, if* $Proj_i(\alpha)$ *satisfies* w_i *for each* $i \in \{1, \ldots, k\}$*, we say that* α satisfies *the set of symbolic traces* $w_1, \ldots, w_k$*. If such a concrete trace exists, we also say that* $w_1, \ldots, w_k$ *is* traceable. □

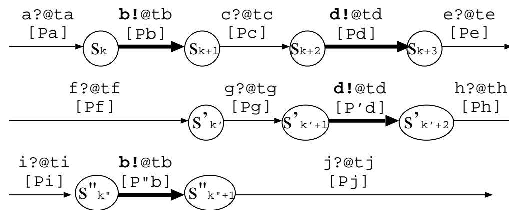


Fig. 3. Symbolic Traces for Concurrent Timed I/O Automata

Sequences on three concurrent timed I/O automata are shown in Fig.3. By definition, actions whose names are the same (output events $b!$ and $d!$ in the figure) are executed synchronously (simultaneously). For deciding the executability of concurrent sequences, we also construct I/O timing intervals from the tail action to the head one of the sequence by similar method described in Section 3.1.

Since tail events of each sequence ($e?$, $h?$ and $j?$) are input events and they are executed independently, a condition which denotes that they are executable independently is $\exists t_e[P_e] \wedge \exists t_h[P_h] \wedge \exists t_j[P_j]$. Assuming that each timing variable of some action (such as t_e) never appears in the transition conditions of other actions (such as P_h or P_j), the expression is equivalent to $\varphi_1 = \exists t_e \exists t_h \exists t_j[[P_e] \wedge [P_h] \wedge [P_j]]$. Next, for synchronous output event $d!$ that is executed most recently before these three input actions, we consider a condition which expresses that "there exists some timing t_d such that these output actions can be executed simultanously and all the succeeding actions can

also be executed.” The condition can be described as $\varphi_2 = \exists t_d[[P_d] \wedge [P'_d] \wedge \varphi_1]$. $e?$, $h?$ and $j?$ are executable if and only if the condition holds. For input events $c?$ and $g?$ which are executed independently, we obtain a condition $\varphi_3 = \exists t_c \exists t_g[[P_c] \wedge [P_g] \wedge \varphi_2]$. We construct conditions for the rest of sequences. For a synchronous output event $b!$, we obtain a condition $\varphi_4 = \exists t_b[[P_b] \wedge [P''_b] \wedge \varphi_3]$. Finally, by considering input events $a?$, $f?$ and $i?$, we obtain a condition $TrCond = \exists t_a \exists t_f \exists t_i[[P_a] \wedge [P_f] \wedge [P_i] \wedge \varphi_4]$. Three sequences shown in Fig.3 are executable simultaneously if and only if $TrCond$ is true.

4 Functional Testing

In this paper, since we consider systems which communicate through networks, each input timing to IUT may have jitter (receive timing of frames and acknowledgment etc.) It is required that there exists some margin for input timing. Furthermore, the test sequence might not be executable for some output timings, since output timings are uncontrollable. Therefore, we apply a static scheduling proposed in [4] in order to obtain suitable I/O timing intervals. By using the scheduling method, for given concurrent sequences we decide whether there exists some execution timing(intervals) for I/O events such that all the sequences are executable. If there exists such timing intervals for sequences, we obtain them and determine the sequences are executable. Although the sequences may not be executed if the IUT does not produce outputs in the expected intervals, we minimize the influence of output timings on the executability of test sequences by making the output timing intervals as wide as possible. Since we can get information about the output timing corresponding to the executability of sequences by applying the scheduling method, we can quickly detect that we fail to execute the given sequences on testing.

4.1 Proposed Testing Method

In the proposed method, we first generate test sequences for checking the correctness of the functions which we intend to test. Then, we derive time intervals that enable the test sequences execute for the I/O events in the generated test sequences.

The problem of deriving time intervals for the I/O events is formally defined as follows.

Definition 9. *Given concurrent timed I/O automata $\mathcal{M} = (M_1, \ldots, M_k)$ and a set of symbolic traces $w_1, \ldots w_k$, the problem of deriving time intervals for I/O events is deriving a set of time intervals $T(w_1), \ldots, T(w_k)$ for the I/O events, which makes all of $w_1, \ldots, w_k$ traceable.* □

For the timing of input event, we can choose several timing covering the derived interval for the input event, when we carry out functional testing actually. In [7], a useful method for choosing adequate input timing for testing multimedia systems has been proposed. Ref. [7] recommends that we should test more at timing close to the boundary in the input timing interval than at timing close to the middle of the interval as shown in Fig.4.

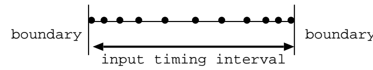


Fig. 4. Input Timing

On the other hand, it is impossible to control the timings of output events. So, in order to make the output timing earlier or later intentionally, we adopt the following way. In [8], we have proposed and implemented a method to execute existing program codes in multi-threaded environments where a given program can be converted into a thread and it can be executed with other threads cooperatively. Here, we assume that each IUT is given as a program (object coeds). By using the method in [8], we can convert a given IUT into a thread as shown in Fig.5. In Fig.5, we add the load-control thread (load controller) as the cooperative thread. If we want to make the output timing later, the tester asks the load-controller to use CPU time more. As the result, the IUT cannot use enough CPU time and the output timing from the IUT becomes late. By adjusting CPU time used by the load-controller, we can make the output timing of the IUT earlier or later intentionally.

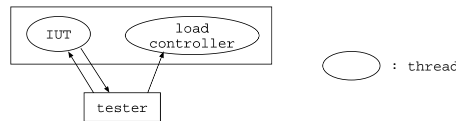


Fig. 5. Testing Environment

Here, we will apply the following functional testing method to the IUT. First, we generate a set of test sequences for functional testing. Then, we derive time intervals which make the given test sequences executable by using the method described in the next sub-section. We give the inputs to the IUT at some timing in the derived time intervals and observe the outputs from the IUT. If the outputs are observed at an earlier time than the expected time intervals, then we make the load-conroller consume much CPU time so that we can observe the outputs at some timing in the derived time intervals. We repeat this functional testing process and give inputs at different timing as shown in Fig.4. If we can observe the corresponding outputs at adequate timing for several times by adjusting the CPU load for the load-controller, we conclude that we succeed the functional testing. If we cannot observe the I/O events at adequate timing even if we repeat the above process, then we conclude that there may have some errors for the implementation of the function to be tested.

4.2 Method for Deriving I/O Timing Intervals

In the proposed method, first, we generate test sequences corresponding to the functions that we want to test and transform them into symbolic traces, which reflect the changes of the values of variables. Based on the scheduling method in [4], we derive executable time intervals for the I/O events in the given traces if they exist.

When we test a synchronized output between audio and video for the media synchronization protocol in Fig.1, we generate test sequences which make the synchronized event "synchronized_play!" executable for three modules (a), (b) and (c), and transform them into symbolic traces. Since module (c) in Fig.1 controls only the synchronization by counting the number of repetation and no time constraint is imposed on the events in it, we omit module (c) from the discussion below. Here, we consider two symbolic traces in Fig.6. These are traces that synchronized_play! is executed synchronously in modules (a) and (b) in Fig.1 when they repeat the cyclic transitions $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_0$ for 4 times and 2 times, respectively.

Hereafter, we propose a method for deriving I/O timing intervals of given test sequences.

sequence in module (a)	sequence in module (b)
audio_receive?@t_{a1}	video_receive?@t_{v1}
audio_decode_request!@t_{a2}	video_decode_request!@t_{v2}
audio_decode_respond?@t_{a3}	video_decode_respond?@t_{v3}
audio_play!@t_{a0}	video_play!@t_{v0}
audio_receive?@t'_{a1}	video_receive?@t'_{v1}
audio_decode_request!@t'_{a2}	video_decode_request!@t'_{v2}
...	video_decode_respond?@t'_{v3}
audio_decode_respond?@t'''_{a3}	**synchronized_play!**@t'_{v0}
synchronized_play!@t'''_{a0}	video_receive?@t''_{v1}
audio_receive?@t''''_{a1}	

Fig. 6. Test Sequences for Synchronized Events (Symbolic Traces)

The condition for each transition $a@t_a$ is supposed to be represented as a logical product of linear inequalities consisting of variable t_a representing execution time of a and variables used in the preceding events. Therefore, the conditions are represented as logical products of expressions whose forms are $\alpha \leq t_a$ or $t_a \leq \beta$. In order to derive the I/O event intervals that make the two traces executable, we give the following constraints.

(1) Each event is never executed ahead of the preceding events. So, the expressions below should be satisfied. We add them as the constraint.

$t_{a1} \leq t_{a2} \leq t_{a3} \leq t_{a0} \leq t'_{a1} \leq \ldots \leq t'''_{a0} \leq t''''_{a1}$
$t_{v1} \leq t_{v2} \leq t_{v3} \leq t_{v0} \leq t'_{v1} \leq \ldots \leq t'_{v0} \leq t''_{v1}$

(2) We introduce two new variables t_{xmin} and t_{xmax} that represent the earliest and latest executable time of t_x, respectively, and replace the original constraint $\alpha \leq t_x \leq \beta$ with $\alpha \leq t_{xmin} \leq t_{xmax} \leq \beta$.

As for the timing of the synchronized event (synchronized_play! in the sequences in Fig.6), we replace the execution time t'''_{a0} and t'_{v0} of two synchronized events with the same variable pair (t_{0min}, t_{0max}). Then the constraint on synchronized_play! in

the audio module is replaced $[t'''_{a3} + 3 \le t_{0min} \le t_{0max} \le t'''_{a3} + 5]$, and that in the video module is replaced $[t'_{v3} + 6 \le t_{0min} \le t_{0max} \le t'_{v3} + 10]$.

(3) As the objective function for solving the above linear programming problem, we give the weighted sum of time intervals $\Sigma w_i(t_{imax} - t_{imin})$, where adequate weights (w_i) are given. If we want to derive wider time intervals for output events, we can give larger weights w_i to the output time intervals.

(4) We solve the linear programming problem.

Thus, the solution tells executable time intervals for each event. For each output event, we judge that the rest of the test sequence is executable if the output is observed in the derived time interval. On the other hand, we decide that given test sequences are not executable if we cannot obtain a solution.

5 Example

We apply our method to two test sequences for checking the correctness of the functions of synchronized events and timeout handling on the media synchronization protocol shown in Fig.1.

5.1 Functional Test Sequences for Synchronized Events

We apply our method to two symbolic traces and obtain the following solutions [2].

$$
\begin{array}{llll}
2.4 \le t_{a1} \le 2.8 & 3.8 \le t_{a2} \le 5.4 & 9.4 \le t_{a3} \le 9.8 & 12.8 \le t_{a0} \le 14.4 \\
16.4 \le t'_{a1} \le 16.8 & 17.8 \le t'_{a2} \le 19.4 & 23.4 \le t'_{a3} \le 23.8 & 26.8 \le t'_{a0} \le 28.4 \\
30.4 \le t''_{a1} \le 30.8 & 31.8 \le t''_{a2} \le 33.4 & 37.4 \le t''_{a3} \le 37.8 & 40.8 \le t''_{a0} \le 42.4 \\
44.4 \le t'''_{a1} \le 44.8 & 45.8 \le t'''_{a2} \le 47.4 & 51.4 \le t'''_{a3} \le 51.8 & 55 \le t_0 \le 56.4 \\
58.4 \le t''''_{a1} \le 59 & & & \\
8 \le t_{v1} \le 9 & 10 \le t_{v2} \le 11 & 20 \le t_{v3} \le 21 & 27 \le t_{v0} \le 28.4 \\
36.4 \le t'_{v1} \le 37 & 38 \le t'_{v2} \le 39.4 & 48.4 \le t'_{v3} \le 49 & 55 \le t_0 \le 56.4 \\
64.4 \le t''_{v1} \le 65 & & &
\end{array}
$$

If we give inputs to IUT at adequate timing in obtained intervals and observe outputs in obtained intervals, the sequence is executable. So we can check the behavior of IUT and test whether IUT synchronizes correctly.

5.2 Functional Test Sequences for Timeout Handling

We consider that if a synchronized event is executed correctly after executing a timeout event, the timeout event (timeout handling) is also executed correctly. We apply our method to the sequences which are obtained by replacing a part of the sequence on module (a) shown in Fig.6 (third appearance of audio_receive?@t''_{a1}, audio_decode_request!@t''_{a2}, audio_decode_respond?@t''_{a3}) with timeout handling (audio_loss!@t''_{a3}). The result is as follows.

---

[2] If we want to derive wider intervals for output events, the interval widths for some input events may become zero. However, it is practically difficult to execute input events at exact timing. To cope with the problem, we specify the minimum interval width for each input event. In this example, for input events audio_receive?, audio_decode_respond?, video_receive? and video_decode_respond?, we specify the minimum widths 0.4,0.3,0.6 and 0.5, respectively.

$$
\begin{array}{cccc}
2 \le t_{a1} \le 2.4 & 3.4 \le t_{a2} \le 5 & 9 \le t_{a3} \le 9.4 & 12.4 \le t_{a0} \le 14 \\
16 \le t'_{a1} \le 16.4 & 17.4 \le t'_{a2} \le 19 & 23 \le t'_{a3} \le 23.3 & 26.3 \le t'_{a0} \le 28 \\
 & & 37 \le t''_{a3} \le 38 & 41 \le t''_{a0} \le 42 \\
44 \le t'''_{a1} \le 45 & 46 \le t'''_{a2} \le 47 & 51 \le t'''_{a3} \le 52 & 55 \le t_0 \le 56 \\
58 \le t''''_{a1} \le 59 & & & \\
8 \le t_{v1} \le 9 & 10 \le t_{v2} \le 11 & 20 \le t_{v3} \le 21 & 27 \le t_{v0} \le 28.4 \\
36.4 \le t'_{v1} \le 37 & 38 \le t'_{v2} \le 39.4 & 48.4 \le t'_{v3} \le 49 & 55 \le t_0 \le 56 \\
64 \le t''_{v1} \le 65 & & &
\end{array}
$$

6 Conclusion

In this paper, we specify a media synchronization protocol as a model in which multiple timed I/O automata work in parallel and cooperatively. Then, we propose a method for functional testing on the model. We also propose a technique for deriving wide executable time intervals of I/O events in a given set of test sequences by using linear programming techniques. As the future work, we are planning to develop an actual environment for functional testing with the multi-threaded programming method in [8].

References

1. R. Alur and D. L. Dill : “A theory of timed automata”, Theoretical Computer Science, Vol. 126, pp.183-235 (1994).
2. T. Higashino, A. Nakata, K. Taniguchi and A. R. Cavalli : “Generating test cases for a timed I/O automaton model”, Proc. of 12th IFIP Workshop on Testing of Communicating Systems (IWTCS’99), pp.197-214 (Sept. 1999).
3. C. M. Huang and C. Wang : “Synchronization for Interactive Multimedia Presentations”, IEEE MULTIMEDIA, Vol. 5, No. 4, pp.44-62 (Oct.-Nov. 1998).
4. H. Katagiri, M. Kirimura, K. Yasumoto and T. Higashino and K. Taniguchi : “Hardware Implementation of Concurrent Periodic EFSMs”, Proc. of Joint International Conference on 13th Formal Description Techniques and 20th Protocol Specification, Testing, and Verification (FORTE/PSTV2000), pp.285-300 (Oct. 2000).
5. D. Lee and M. Yannakakis : “Principles and Methods of Testing Finite State Machines - A Survey”, Proc. of the IEEE, Vol. 84, No. 8 (1996).
6. T. D. C. Little and A. Ghafoor : “Synchronization and storage models for multimedia objects”, IEEE Journal of Selected Areas in Communications, Vol. 8, No. 3, pp.413-427 (Apr. 1990).
7. V. Misic, S. T. Chanson and S. C. Cheung : “Towards a Framework for Testing Distributed Multimedia Software Systems”, Proc. International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE98) (Apr. 1998).
8. K. Abe, T. Matsuura, K. Yasumoto and T. Higashino : “A Method to Execute Existing Program Codes in Multi-theread Environments and Its Implementation” Journal of Information Processing Society of Japan, Vol. 41, No. 9, pp.2603-2613 (Sep. 2000) (In Japanese).